A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage

James S. Plank Jianqiang Luo Catherine D. Schuman Lihao Xu
Zooko Wilcox-O'Hearn

Abstract

Over the past five years, large-scale storage installations have required fault-protection beyond RAID-5, leading to a flurry of research on and development of erasure codes for multiple disk failures. Numerous open-source implementations of various coding techniques are available to the general public. In this paper, we perform a head-to-head comparison of these implementations in encoding and decoding scenarios. Our goals are to compare codes and implementations, to discern whether theory matches practice, and to demonstrate how parameter selection, especially as it concerns memory, has a significant impact on a code's performance. Additional benefits are to give storage system designers an idea of what to expect in terms of coding performance when designing their storage systems, and to identify the places where further erasure coding research can have the most impact.

1 Introduction

In recent years, erasure codes have moved to the fore to prevent data loss in storage systems composed of multiple disks. Storage companies such as Allmydata [1], Cleversafe [7], Data Domain [34], Network Appliance [20] and Panasas [30] are all delivering products that use erasure codes beyond RAID-5 for data availability. Academic projects such as LoCI [3], Oceanstore [27], and Pergamum [29] are doing the same. And of equal importance, major technology corporations such as Hewlett Packard [32], IBM [11, 12] and Microsoft [14, 15] are performing active research on erasure codes for storage systems.

Along with proprietary implementations of erasure codes, there have been numerous open source implementations of a variety of erasure codes that are available for download [7, 18, 21, 22, 31]. The intent of most of these projects is to provide storage system developers with high quality tools. As such, there is a need to understand how these codes and implementations perform.

In this paper, we compare the encoding and decoding performance of six open-source implementations of five different types of erasure codes: Classic Reed-Solomon codes [26], Cauchy Reed-Solomon codes [6], EVENODD [4], Row Diagonal Parity (RDP) [8] and Minimal Density RAID-6 codes [5, 24, 23]. The latter three codes are specific to RAID-6 systems that can tolerate exactly two failures. Our exploration seeks not only to compare codes to but also to understand which features and parameters lead to good coding performance.

We summarize the main results as follows:

- The special-purpose RAID-6 codes vastly outperform their general-purpose counterparts, with RDP leading the field.
- Cauchy Reed-Solomon coding outperforms classic Reed-Solomon coding significantly, as long as attention is paid to generating good encoding matrices.
- An optimization called *Code-Specific Hybrid Reconstruction* [13] is necessary to achieve good decoding speeds in many of the codes.
- Parameter selection can have a huge impact on how well an implementation performs. Not only must the number of computational operations be considered, but also how the code interacts with the memory hierarchy, especially the caches.
- There is a need to achieve the types improvements that the RAID-6 codes show for higher numbers of failures.

Of the six libraries tested, *Zfec* [31] implemented the fastest classic Reed-Solomon coding, and *Jerasure* [22] implemented the fastest versions of the others.

2 Nomenclature and Erasure Codes

It is an unfortunate consequence of the history of erasure coding research that there is no unified nomenclature for erasure coding. We borrow terminology mostly from Hafner *et al* [13], but try to conform to more classic coding terminology (e.g. [5, 19]) when appropriate.

Our storage system is composed of an **array** of n disks, each of which is the same size. Of these n disks, k of them hold **data** and the remaining m hold **coding** information, often termed **parity**, which is calculated from the data. We label the data disks $D_0, \ldots, D_{k-1}$ and the parity disks $C_0, \ldots, C_{m-1}$. A typical system is pictured in Figure 1.

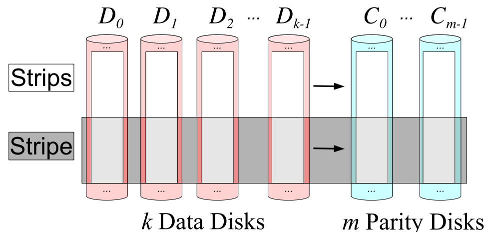


Figure 1: A typical storage system with erasure coding.

We are concerned with **Maximum Distance Separable (MDS)** codes, which have the property that if any m disks fail, the original data may be reconstructed [19]. When encoding, one partitions each disk into **strips** of a fixed size. Each parity strip is encoded using one strip from each data disk, and the collection of $k+m$ strips that encode together is called a **stripe**. Thus, as in Figure 1, one may view each disk as a collection of strips, and one may view the entire system as a collection of stripes. Note that stripes are each encoded independently, and therefore if one desires to rotate the data and parity among the n disks for load balancing, one may do so by switching the disks' identities for each stripe.

2.1 Reed-Solomon (RS) Codes

Reed-Solomon codes [26] have the longest history. The strip unit is a w-bit word, where w must be large enough that $n \leq 2^w + 1$. So that words may be manipulated efficiently, w is typically constrained so that words fall on machine word boundaries: $w \in \{8, 16, 32, 64\}$. However, as long as $n \leq 2^w + 1$, the value of w may be chosen at the discretion of the user. Reed-Solomon codes treat each word as a number between 0 and $2^w - 1$, and operate on these numbers with *Galois Field* arithmetic ($GF(2^w)$), which defines addition, multiplication and division on these words such that the system is closed and well-behaved [19].

The act of encoding with Reed-Solomon codes is simple linear algebra. A *Generator Matrix* is constructed from a *Vandermonde* matrix, and this matrix is multiplied by the k data words to create a *codeword* composed of the k data and m coding words. We picture the process in Figure 2 (note, we draw the transpose of the Generator Matrix to make the picture clearer).

Figure 2: Reed-Solomon coding for $k = 4$ and $m = 2$. Each element is a number between 0 and $2^w - 1$.

When disks fail, one decodes by deleting rows of G^T, inverting it, and multiplying the inverse by the surviving words. This process is equivalent to solving a set of independent linear equations. The construction of G^T from the Vandermonde matrix ensures that the matrix inversion is always successful.

In $GF(2^w)$, addition is equivalent to bitwise exclusive-or (XOR), and multiplication is more complex, typically implemented with multiplication tables or discrete logarithm tables. For this reason, Reed-Solomon codes are considered expensive. There are several open-source implementations of RS coding, which we detail in Section 3.

2.2 Cauchy Reed-Solomon (CRS) Codes

CRS codes [6] modify RS codes in two ways. First, they define a different construction of the Generator matrix using Cauchy matrices instead of Vandermonde matrices. Second, they eliminate the expensive multiplications of RS codes by converting them to extra XOR operations. Note, this second modification can apply to Vandermonde-based RS codes as

well. This modification transforms G^T from a $n \times k$ matrix of w-bit words to a $wn \times wk$ matrix of bits. As with RS coding, w must be selected so that $n \leq 2^w+1$.

Instead of operating on single words, CRS coding operates on entire strips. In particular, strips are partitioned into w *packets*, and these packets may be large. The act of coding now involves only XOR operations – a coding packet is constructed as the XOR of all data packets that have one bits in the coding packet's row of G^T. The process is depicted in Figure 3, which illustrates how the last coding packet is created as the XOR of the six data packets identified by the last row of G^T.

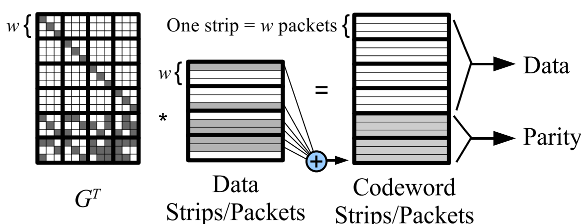


Figure 3: CRS example for $k = 4$ and $m = 2$.

To make XORs efficient, the packet size must be a multiple of the machine's word size. The strip size is therefore equal to w times the packet size. Since w no longer relates to the machine word sizes, w is no longer constrained to $[8, 16, 32, 64]$; instead, any value of w may be selected as long as $n \leq 2^w$.

Decoding in CRS is analogous to RS coding — all rows of G^T corresponding to failed packets are deleted, and the matrix is inverted and employed to recalculate the lost data.

Since the performance of CRS coding is directly related to the number of ones in G^T, there has been research on constructing Cauchy matrices that have fewer ones than the original CRS constructions [25]. The *Jerasure* library [22] uses additional matrix transformations to improve these matrices further. Additionally, in the restricted case when $m = 2$, the *Jerasure* library uses results of a previous enumeration of all Cauchy matrices to employ provably optimal matrices for all $w \leq 32$.

2.3 EVENODD and RDP

EVENODD [4] and RDP [8] are two codes developed for the special case of RAID-6, which is when $m = 2$. Conventionally in RAID-6, the first parity drive is labeled P, and second is labeled Q. The P drive is equivalent to the parity drive in a RAID-4 system, and the Q drive is defined by parity equations that have distinct patterns.

Although their original specifications use different terms, EVENODD and RDP fit the same paradigm as CRS coding, with strips being composed of w packets. In EVENODD, w is constrained such that $k + 1 \leq w$ and $w + 1$ is a prime number. In RDP, $w + 1$ must be prime and $k \leq w$. Both codes perform the best when $(w - k)$ is minimized. In particular, RDP achieves optimal encoding and decoding performance of $(k-1)$ XOR operations per coding word when $k = w$ or $k + 1 = w$. Both codes' performance decreases as $(w - k)$ increases.

2.4 Minimal Density RAID-6 Codes

If we encode using a Generator bit-matrix for RAID-6, the matrix is quite constrained. In particular, the first kw rows of G^T compose an identity matrix, and in order for the P drive to be straight parity, the next w rows must contain k identity matrices. The only flexibility in a RAID-6 specification is the composition of the last w rows. In [5], Blaum and Roth demonstrate that when $k \leq w$, these remaining w rows must have at least $kw + k - 1$ ones for the code to be MDS. We term MDS matrices that achieve this lower bound *Minimal Density* codes.

There are three different constructions of Minimal Density codes for different values of w:

- **Blaum-Roth** codes when $w + 1$ is prime [5].
- **Liberation** codes when w is prime [24].
- The **Liber8tion** code when $w = 8$ [23].

These codes share the same performance characteristics. They encode with $(k - 1) + \frac{k-1}{2w}$ XOR operations per coding word. Thus, they perform better when w is large, achieving asymptotic optimality as $w \to \infty$. Their decoding performance is slightly worse, and requires an technique called *Code-Specific Hybrid Reconstruction* [13] to achieve near-optimal performance [24].

The Minimal Density codes also achieve near-optimal *updating* performance when individual pieces of data are modified [25]. This performance is significantly better than EVENODD and RDP, which are worse by a factor of roughly 1.5 [24].

2.5 Anvin's RAID-6 Optimization

In 2007, Anvin posted an optmziation of RS encoding for RAID-6 [2]. For this optimization, the P row of G^T contains all ones so that the P drive may be parity. The Q row contains the number 2^i in $GF(2^w)$ in column i (zero-indexed) so that the contents of the Q drive may be calculated by successively XORing drive i's data into the Q drive and multiplying that sum by two. Since multiplication by two may be implemented much faster than general multiplication in $GF(2^w)$, this optimizes the performance of encoding over standard RS implementations. Decoding remains unoptimized.

3 Open Source Libraries

We test six open source erasure coding libraries. These are all freely available libraries from various sources on the Internet, and range from brief proofs of concept (e.g. *Luby*) to tuned and supported code intended for use in real systems (e.g. *Zfec*). We present them chronologically.

Luby: CRS coding was developed at the ISCI lab in Berkeley, CA in the mid 1990's [6]. The authors released a C version of their codes in 1997, which is available from ISCI's web site [18]. The library supports all settings of k, m, w and packet sizes. The matrices employ the original constructions from [6], which are not optimized to minimize the number of ones.

Schifra: In July, 2000, the first version of the *Schifra* coding library was made avaiable for free download. The library is written in C++, implementing RS coding with $w = 8$, with a robust API and support [21]. There is a license for documentation of the library and for a high-performance version. Thus, the version we have tested does not represent the best performance of *Schifra*. However, it does represent the performance a developer can expect from the freely available download.

Zfec: The *Zfec* library for erasure coding has been in development since 2007, but its roots have been around for over a decade. *Zfec* is built on top of a RS coding library developed for reliable multicast by Rizzo [28]. That library was based on previous work by Karn *et al* [17], and has seen wide use and tuning. *Zfec* is based on Vandermonde matrices when $w = 8$. The latest version (1.4.0) was posted in January, 2008 [31]. The library is programmable, portable and actively supported by the author. It includes command-line tools and APIs in C, Python and Haskell.

Jerasure: Jerasure is a C library released in 2007 that supports a wide variety of erasure codes, including RS coding, CRS coding, general Generator matrix and bit-matrix coding, and Minimal Density RAID-6 coding [22]. RS coding may be based on Vandermonde or Cauchy matrices, and w may be 8, 16 or 32. Anvin's optimization is included for RAID-6 applications. For CRS coding, *Jerasure* employs provably optimal encoding matrices for RAID-6, and constructs optimized matrices for larger values of m. Additionally, the three Minimal Density RAID-6 codes are supported. To improve performance of the bit-matrix codes, especially the decoding performance, the Code-Specific Hybrid Reconstruction optimization [13] is included. *Jerasure* is released under the GNU LGPL.

Cleversafe: In May, 2008, Cleversafe exported the first open source version of its dispersed storage system [7]. Written entirely in Java, it supports the same API as Cleversafe's proprietary system, which is notable as one of the first commercial distributed storage systems to implement availability beyond RAID-6. For this paper, we obtained a version containing just the the erasure coding part of the open source distribution. It is based on Luby's original CRS implementation [18] with $w = 8$.

EVENODD/RDP: There are no open source versions of EVENODD or RDP coding. However, RDP may be implemented as a bit-matrix code, which, when combined with Code-Specific Hybrid Reconstruction yields the same performance as the original specification of the code [15]. EVENODD may also be implemented with a bit-matrix whose operations may be scheduled to achieve the code's original performance [15]. We use these observations to implement both codes as bit-matrices with tuned schedules in *Jerasure*. Since EVENODD and RDP codes are patented, this implementation is not available to the public, as its sole intent is for performance comparison.

4 Encoding Experiment

We perform two sets of experiments – one for encoding and one for decoding. For the encoding experiment, we seek to measure the performance of taking a large data file and splitting and encoding it into $n = k + m$ pieces, each of which will reside on a different disk, making the system tolerant to up

to m disk failures. Our encoder thus reads a data file, encodes it, and writes it to $k+m$ data/coding files, measuring the performance of the encoding operations.

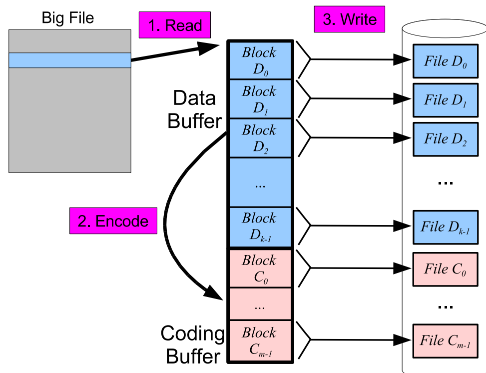


Figure 4: The encoder utilizes a data buffer and a coding buffer to encode a large file in stages.

Since memory utilization is a concern, and since large files exceed the capacity of most computers' memories, our encoder employs two fixed-size buffers, a *Data Buffer* partitioned into k *blocks* and a *Coding Buffer* partitioned into m blocks. The encoder reads an entire data buffer's worth of data from the big file, encodes it into the coding buffer and then writes the contents of both buffers to $k+m$ separate files. It repeats this process until the file is totally encoded, recording both the total time and the encoding time. The high level process is pictured in Figure 4.

The blocks of the buffer are each partitioned into s strips, and each strip is partitioned either into words of size w (RS coding, where $w \in \{8, 16, 32, 64\}$), or into w packets of a fixed size **PS** (all other codes – recall Figure 3). To be specific, each block D_i (and C_j) is partitioned into strips $DS_{i,0}, \ldots, DS_{i,s-1}$. (and $CS_{j,0}, \ldots, CS_{j,s-1}$), each of size w**PS**. Thus, the data and coding buffer sizes are dependent on the various parameters. Specifically, the data buffer size equals (ksw**PS**) and the coding buffer size equals (msw**PS**).

Encoding is done on a stripe-by-stripe basis. First, the data strips $DS_{0,0}, \ldots, DS_{k-1,0}$ are encoded into the coding strips $CS_{0,0}, \ldots, CS_{m-1,0}$. This achieves the encoding of stripe 0, pictured in Figure 5. Each of the s stripes is successively encoded in this manner.

Thus, there are myriad parameters that the encoder allows the user to set. These are k, m, w (subject to the code's constraints), s and **PS**. When we

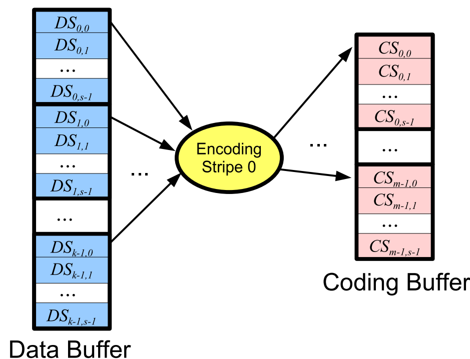


Figure 5: How the data and coding buffers are partitioned, and the encoding of Stripe 0 from data strips $DS_{0,0}, \ldots, DS_{k-1,0}$ into coding strips $CS_{0,0}, \ldots, CS_{m-1,0}$.

mention setting the buffer size below, we are referring to the size of the data buffer, which is (ksw**PS**).

4.1 Machines for Experimentation

We employed two machines for experimentation. Neither is exceptionally high-end, but each represents middle-range commodity processors, which should be able to encode and decode comfortably within I/O speed limits. The first is a Macbook with a 32-bit 2GHz Intel Core Duo processor, with 1GB of RAM, a L1 cache of 32KB and a L2 cache of 2MB. Although the machine has two cores, the encoder only utilizes one. The operating system is Mac OS X, version 10.4.11, and the encoder is executed in user space while no other user programs are being executed. As a baseline, we recorded a **memcpy()** speed of 6.13 GB/sec and an **XOR** speed of 2.43 GB/sec.

The second machine is a Dell workstation with an Intel Pentium 4 CPU running at 1.5Ghz with 1GB of RAM, an 8KB L1 cache and a 256KB L2 cache. The operating system is Debian GNU Linux revision 2.6.8-2-686, and the machine is a 32-bit machine. The **memcpy()** speed is 2.92 GB/sec and the **XOR** speed is 1.32 GB/sec.

4.2 Encoding with Large Files

Our intent was to measure the actual performance of encoding a large video file. However, doing large amounts of I/O causes a great deal of variability in performance timings. We exemplify with Figure 6. This data is from the Macbook, where we use a 256

MB video file for input. The encoder works as described in Section 4 with $k = 10$ and $m = 6$. However, we perform no real encoding. Instead we simply zero the bytes of the coding buffer before writing it to disk. In the figure, we modify the size of the data buffer from a small size of 64 KB to 256 MB – the size of the video file itself.

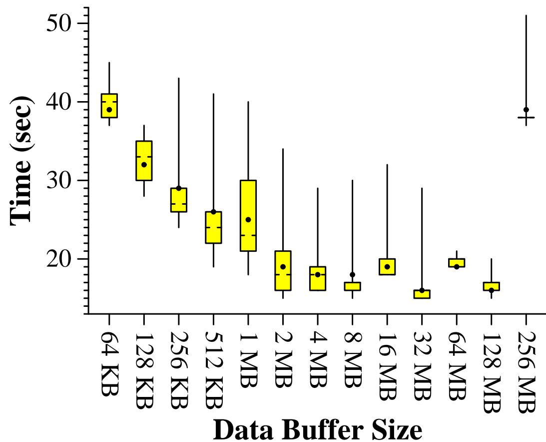


Figure 6: Times to read a 256 MB video, peform a dummy encoding when $k = 10$ and $m = 6$, and write to 16 data/coding files.

In Figure 6, each data point is the result of ten runs executed in random order. A tukey plot is given, which has bars to the maximum and minimum values, a box encompassing the first to the third quartile, hash marks at the median and a dot at the mean. While there is a clear trend toward improving performance as the data buffer grows to 128 MB, the variability in performance is colossal: between 15 and 20 seconds for many runs. Running Unix's **split** utility on the file reveals similar variability.

Because of this variability, the tests that follow remove the I/O from the encoder. Instead, we simulate reading by filling the buffer with random bytes, and we simulate writing by zeroing the buffers. This reduces the variability of the runs tremendously – the results that follow are all averages of over 10 runs, whose maximum and minimum values differ by less than 0.5 percent. The encoder measures the times of all coding activites using Unix's **gettimeofday()**. To confirm that these times are accurate, we also subtracted the wall clock time of a dummy control from the wall clock time of the encoder, and the two matched to within one percent.

Figure 6 suggests that the size of the data buffer can impact performance, although it is unclear whether the impact comes from memory effects or from the file system. To explore this, we performed a second set of tests that modify the size of the data buffer while performing a dummy encoding. We do not graph the results, but they show that with the I/O removed, the effects of modifying the buffer size are negligible. Thus, in the results that follow, we maintain a data buffer size of roughly 100 KB. Since actual buffer sizes depend on k, m, w and **PS**, they cannot be affixed to a constant value; instead, they are chosen to be in the ballpark of 100 KB. This is large enough to support efficient I/O, but not so large that it consumes all of a machine's memory, since in real systems the processors may be multitasking.

4.3 Parameter Space

We test four combinations of k and m – we will denote them by $[k, m]$. Two combinations are RAID-6 scenarios: [6,2] and [14,2]. The other two represent 16-disk stripes with more fault-tolerance: [12,4] and [10,6]. We chose these combinations because they represent values that are likely to be seen in actual usage. Although large and wide-area storage installations are composed of much larger numbers of disks, the stripe sizes tend to stay within this medium range, because the benefits of large stripe sizes show diminishing returns compared to the penalty of extra coding overhead in terms of encoding performance and memory use. For example, Cleversafe's widely dispersed storage system uses [10,6] as its default [7]; Allmydata's archival online backup system uses [3,7], and both Panasas [30] and Pergamum [29] report keeping their stripe sizes at or under 16.

For each code and implementation, we test its performance by encoding a randomly generated file that is 1 GB in size. We test all legal values of $w \leq 32$. This results in the following tests.

- *Schifra*: RS coding, $w = 8$ for all combinations of $[k, m]$.
- *Zfec*: RS coding, $w = 8$ for all combinations of $[k, m]$.
- *Luby*: CRS coding, $w \in \{4, \ldots, 12\}$ for all combinations of $[k, m]$, and $w = 3$ for [6,2].
- *Cleversafe*: CRS coding, $w = 8$ for all combinations of $[k, m]$.
- *Jerasure*:

- RS coding, $w \in \{8, 16, 32\}$ for all combinations of $[k, m]$. Anvin's optimization is included for the RAID-6 tests.
- CRS coding, $w \in \{4, \ldots, 32\}$ for all combinations of $[k, m]$, and $w = 3$ for [6,2].
- Blaum-Roth codes, $w \in \{6, 10, 12\}$ for [6,2] and $w \in \{16, 18, 22, 28, 30\}$ for [6,2] and [14,2].
- Liberation codes, $w \in \{7, 11, 13\}$ for [6,2] and $w \in \{17, 19, 23, 29, 31\}$ for [6,2] and [14,2].
- The Liber8tion code, $w = 8$ for [6,2].

- *EVENODD*: Same parameters as Blaum-Roth codes in *Jerasure* above.
- *RDP*: Same parameters as *EVENODD*.

4.4 Impact of the Packet Size

Our experience with erasure coding led us to experiment first with modifying the packet sizes of the encoder. There is a clear tradeoff: lower packet sizes have less tight XOR loops, but better cache behavior. Higher packet sizes perform XORs over larger regions, but will cause more cache misses. To exemplify a typical example, consider Figure 7, which shows the performance of RDP on the [6,2] configuration when $w = 6$, on the Macbook. We test every packet size from 4 to 10000 and display the speed of encoding.

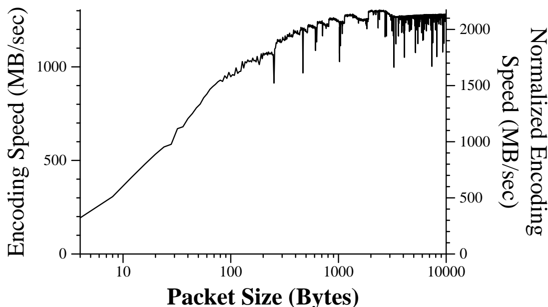


Figure 7: The effect of modifying the packet size on RDP coding, $k = 6$, $m = 2$, $w = 6$ on the Macbook.

We display two y-axes. On the left is the encoding speed. This is the size of the input file divided by the time spent encoding and is the most natural metric to plot. On the right, we normalize the encoding speed so that we may compare the performance of encoding across configurations. The normalized encoding speed is calculated as:

$$\frac{(\text{Encoding Speed})\ m(k-1)}{k}. \qquad (1)$$

This is derived as follows. Let S be the file's size and t be the time to encode. The file is split and encoded into $m+k$ files, each of size $\frac{S}{k}$. The encoding itself creates $\frac{Sm}{k}$ bytes worth of data, and therefore the speed per coding byte is $\frac{Sm}{kt}$. Optimal encoding takes $k-1$ XOR operations per coding drive [33]; therefore we can normalize the speed by dividing the time by $k-1$, leaving us with $\frac{Sm(k-1)}{kt}$, or Equation 1 for the normalized encoding speed.

The shape of this curve is typical for all codes on both machines. In general, higher packet sizes perform better than lower ones; however there is a maximum performance point which is achieved when the code makes best use of the L1 cache. In this test, the optimal packet size is 2400 bytes, achieving a normalized encoding speed of 2172 MB/sec. Unfortunately, this curve does not monotonically increase to or decrease from its optimal value. Worse, there can be radical dips in performance between adjacent packet sizes, due to collisions between cache entries. For example, at packet sizes 7732, 7736 and 7740, the normalized encoding speeds are 2133, 2066 and 2129 MB/sec respectively.[1]

We do not attempt to find the optimal packet sizes for each of the codes. Instead, we perform a search algorithm that works as follows. We test a *region* r of packet sizes by testing each packet size from r to $r+36$ (packet sizes must be a multiple of 4). We set the region's performance to be the average of the five best tests. To start our search, we test all regions that are powers of two from 64 to 32K. We then iterate, finding the best region r, and then testing the two regions that are halfway between the two values of r that we have tested that are adjacent to r. We do this until there are no more regions to test, and select the packet size of all tested that performed the best. For example, the search for the RDP instance of Figure 7 tested only 202 packet sizes (as opposed to 2500 to generate Figure 7) to arrive at a packet size of 2588 bytes, which encodes at a normalized speed of 2164 MB/sec (0.3% worse than the best packet size of 2400 bytes).

[1] We reiterate that each data point in our graphs represents over 10 runs, and the repetitions are consistent to within 0.5 percent.

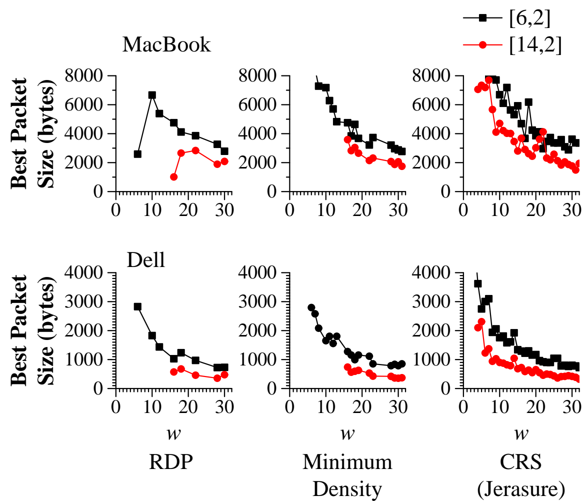


Figure 8: The effect of modifying w on the best packet sizes found.

One expects the optimal packet size to decrease as k, m and w increase, because each of these increases the stripe size. Thus smaller packets are necessary for most of the stripe to fit into cache. We explore this effect in Figure 8, where we show the best packet sizes found for different sets of codes – RDP, Minimum Density, and *Jerasure*'s CRS – in the two RAID-6 configurations. For each code, the larger value of k results in a smaller packet size, and as a rough trend, as w increases, the best packet size decreases.

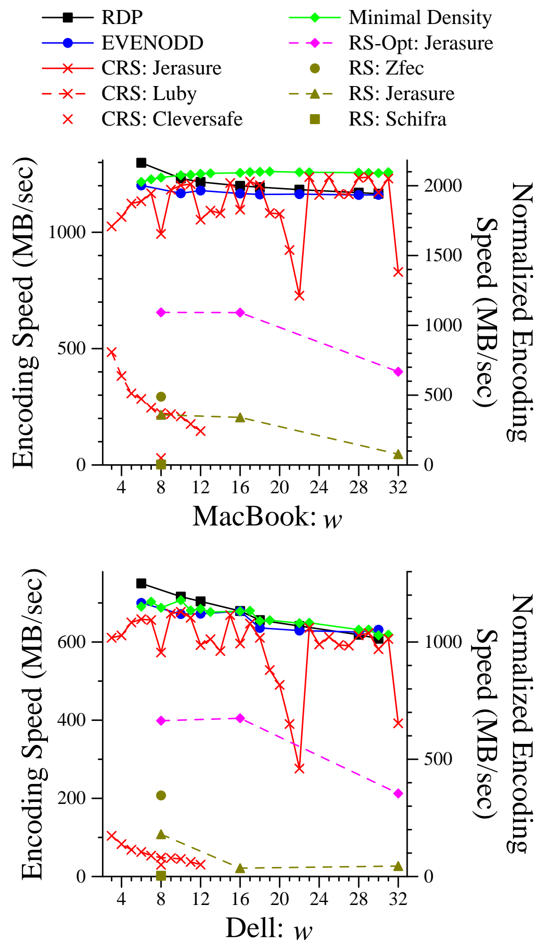


Figure 9: Encoding performance for [6,2].

4.5 Overall Encoding Performance

We now present the performance of each of the codes and implementations. In the codes that allow a packet size to be set, we select the best packet size from the above search. The results for the [6,2] configuration are in Figure 9.

Although the graphs for both machines appear similar, there are interesting features of both. We concentrate first on the MacBook. The specialized RAID-6 codes outperform all others, with RDP's performance with $w = 6$ performing the best. This result is expected, as RDP achieves optimal performance when $k = w$.

The performance of these codes is typically quantified by the number of XOR operations performed [5, 4, 8, 24, 23]. To measure how well number of XORs matches actual performance, we present the number of gigabytes XOR'd by each code in Figure 10.

On the MacBook, the number of XORs is an excellent indicator of performance, with a few exceptions (CRS codes for $w \in \{21, 22, 32\}$). As predicted by XOR count, RDP's performance suffers as w increases, while the Minimal Density codes show better performance. Of the three special-purpose RAID-6 codes, EVENODD performs the worst, although the margins are not large (the worst performing EVENODD encodes at 89% of the speed of the best RDP).

The performance of *Jerasure*'s implementation the CRS codes is also excellent, although the choice of w is very important. The number of ones in the CRS generator matrices depends on the number of bits in the Galois Field's primitive polynomial. The polynomials for $w \in \{8, 12, 13, 14, 16, 19, 24, 26, 27, 30, 32\}$ have one more bit than the others, resulting in worse performance. This is important, as $w \in \{8, 16, 32\}$ are very natural choices since they allow strip sizes to be powers of two.

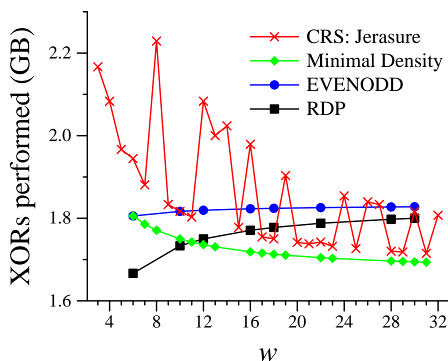


Figure 10: Gigabytes XOR'd by each code in the [6,2] tests. The number of XORs is independent of the machine used.

The *Luby* and *Cleversafe* implementations of CRS coding perform much worse than *Jerasure*. There are several reasons for this. First, they do not optimize the generator matrix in terms of number of ones, and thus perform many more XOR operations, from 3.2 GB of XORs when $w = 3$ to 13.5 GB when $w = 12$. Second, both codes use a dense, bit-packed representation of the generator matrix, which means that they spend quite a bit of time performing bit operations to check matrix entries, many of which are zeros and could be omitted. *Jerasure* converts the matrix to a schedule which eliminates all of the matrix traversal and entry checking during encoding. *Cleversafe's* poor performance relative to Luby can most likely be attributed to the Java implementation and the fact that the packet size is hard coded to be very small (since Cleversafe routinely distributes strips in units of 1K).

Of the RS implementations, the implementation tailored for RAID-6 (labeled "RS-Opt") performs at a much higher rate than the others. This is due to the fact that it does not perform general-purpose Galois Field multiplication over w-bit words, but instead performs a machine word's worth of multiplication by two at a time. Its performance is better when $w \leq 16$, which is not a limitation as $w = 16$ can handle a system with a total of 64K drives. The *Zfec* implementation of RS coding outperforms the others. This is due to the heavily tuned implementation, which performs explicit loop unrolling and hard-wires many features of $GF(2^8)$ which the other libraries do not. Both *Zfec* and *Jerasure* use precomputed multiplication and division tables for $GF(2^8)$. For $w = 16$, *Jerasure* uses discrete logarithms, and for $w = 32$, it uses a recursive table-lookup scheme.

The results on the Dell are similar to the MacBook with some significant differences. The first is that larger values of w perform worse relative to smaller values, regardless of their XOR counts. While the Minimum Density codes eventually outperform RDP for larger w, their overall performance is far worse than the best performing RDP instance. For example, Liberation's encoding speed when $w = 31$ is 82% of RDP's speed when $w = 6$, as opposed to 97% on the MacBook. We suspect that the reason for this is the smaller L1 cache on the Dell, which penalizes the strip sizes of the larger w.

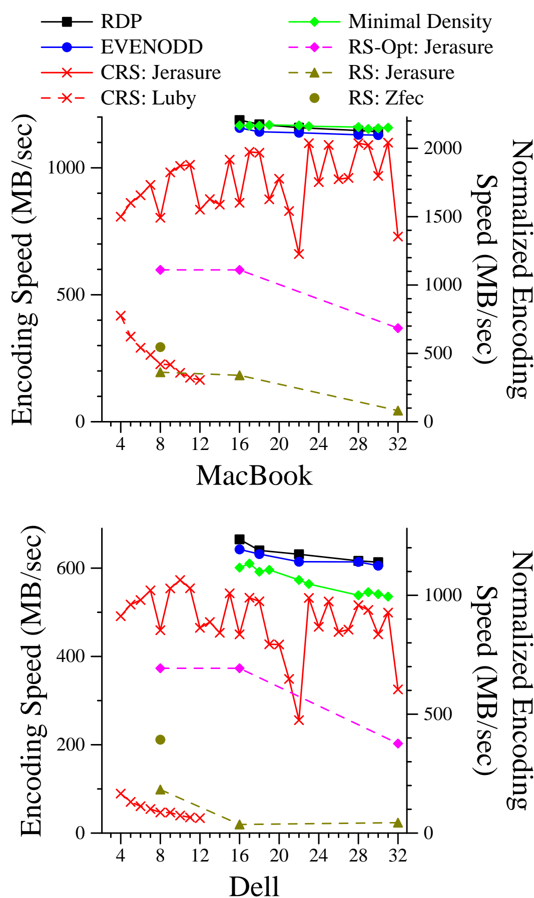


Figure 11: Encoding performance for [14,2].

The final difference between the MacBook and the Dell is that *Jerasure*'s RS performance for $w = 16$ is much worse than for $w = 8$. We suspect that this is because *Jerasure*'s logarithm tables are not optimized for space, consuming 1.5 MB of memory, since there are six tables of 256 KB each [22]. The lower bound

is two 128 KB tables, which should exhibit better behavior on the Dell's limited cache.

Figure 11 displays the results for [14,2] (we omit *Cleversafe* and *Shifra*, since their performance is so much worse than the others). The trends are similar to [6,2], with the exception that on the Dell, the Minimum Density codes perform significantly worse than RDP and EVENODD, even though their XOR counts follow the performance of the MacBook. The definition of the normalized encoding speed means that if a code is encoding optimally, its normalized encoding speed should match the XOR speed. In both machines, RDP's [14,2] normalized encoding speed comes closest to the measured XOR speed, meaning that in implementation as in theory, this is an extremely efficient code.

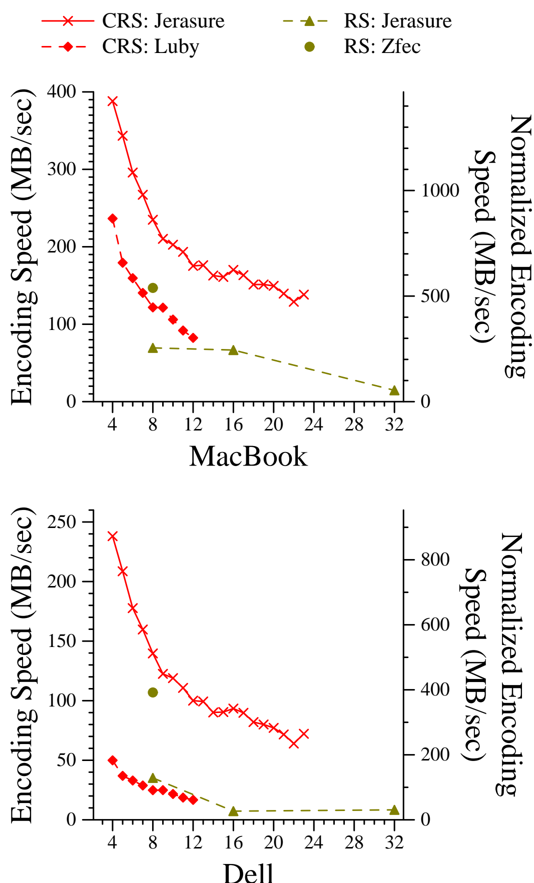


Figure 12: Encoding performance for [12,4].

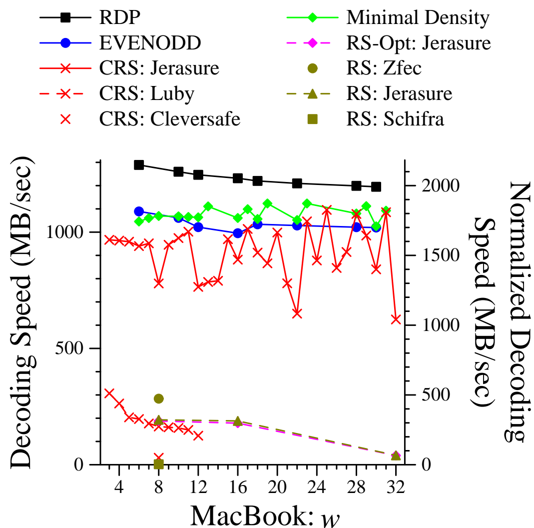
Figure 12 displays the results for [12,4]. Since this is no longer a RAID-6 scenario, only the RS and CRS codes are displayed. The normalized performance of *Jerasure's* CRS coding is much worse now because the generator matrices are more dense and cannot be optimized as they can when $m = 2$. As such, the codes perform more XOR operations than when $k = 14$. For example, when $w = 4$ *Jerasure's* CRS implementation performs 17.88 XORs per coding word; optimal is 11. This is why the normalized coding speed is much slower than in the best RAID-6 cases. Since *Luby's* code does not optimize the generator matrix, it performs more XORs (23.5 per word, as opposed to 17.88 for *Jerasure*), and as a result is slower.

The RS codes show the same performance as in the other tests. In particular, *Zfec's* normalized performance is roughly the same in all cases. For space purposes, we omit the [10,6] results as they show the same trends as the [12,4] case. The peak performer is *Jerasure's* CRS, achieving a normalized speed of 1409 MB/sec on the MacBook and 869.4 MB/sec on the Dell. *Zfec's* normalized encoding speeds are similar to the others: 528.4 MB/sec on the MacBook and 380.2 MB/sec on the Dell.

5 Decoding Performance

To test the performance of decoding, we converted the encoder program to perform decoding as well. Specifically, the decoder chooses m random data drives, and then after each encoding iteration, it zeros the buffers for those drives and decodes. We only decode data drives for two reasons. First, it represents the hardest decoding case, since all of the coding information must be used. Second, all of the libraries except *Jerasure* decode only the data, and do not allow for individual coding strips to be re-encoded without reencoding all of them. While we could have modified those libraries to re-encode individually, we did not feel that it was in the spirit of the evaluation. Before testing, we wrote code to double-check that the erased data was decoded correctly, and in all cases it was.

We show the performance of two configurations: [6,2] in Figure 13 and [12,4] in Figure 14. The results are best viewed in comparison to Figures 9 and 12. The results on the MacBook tend to match theory. RDP decodes as it encodes, and the two sets of speeds match very closely. EVENODD and the Minimal Density codes both have slightly more complexity in decoding, which is reflected in the graph. As mentioned in [23], the Minimal Density codes benefit greatly from Code-Specific Hybrid Reconstruction [13], which is implemented in *Jerasure*. Without the optimization, the decoding performance of



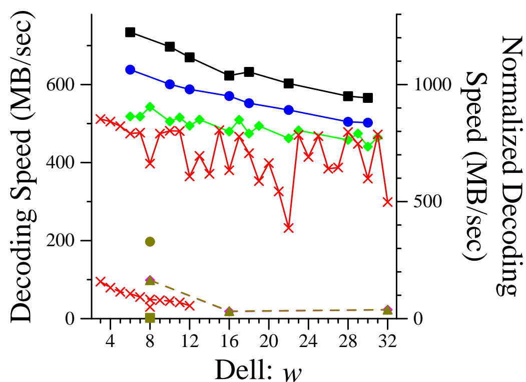


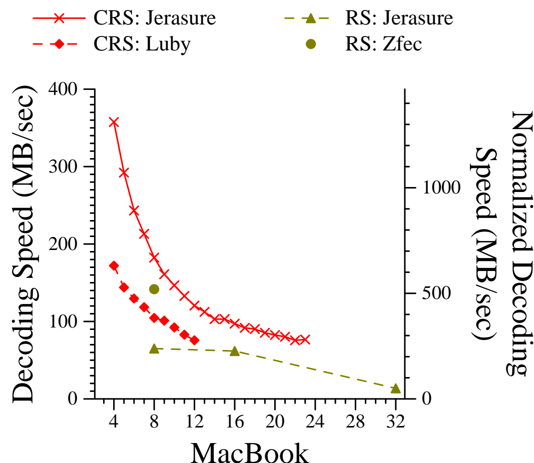
Figure 13: Decoding performance for [6,2].

these codes would be unacceptable. For example, in the [6,2] configuration on the MacBook, the Liberation code for $w = 31$ decodes at a normalized rate of 1820 MB/sec. Without Code-Specific Hybrid Reconstruction, the rate is a factor of six slower: 302.7 MB/sec. CRS coding also benefits from the optimization. Again, using an example where $w = 31$, normalized speed with the optimization is 1809 MB/s, and without it is 261.5 MB/sec.

The RS decoders perform identically to their encoding counterparts with the exception of the RAID-6 optimized version. This is because the optimization applies only to encoding and defaults to standard RS decoding. Since the only difference between RS encoding and decoding is the inversion of a $k \times k$ matrix, the fact that encoding and decoding performance match is expected.

On the Dell, the trends between the various codes follow the encoding tests. In particular, larger values of w are penalized more by the small cache.

In the [12,4] tests, the performance trends of the CRS codes are the same, although the decoding proceeds more slowly. This is more pronounced in *Jerasure*'s implementation than in *Luby's*, and can be explained by XORs. In *Jerasure*, the program attempts to minimize the number of ones in the encoding matrix, without regard to the decoding matrix. For example, when $w = 4$, CRS encoding requires 5.96 GB of XORs. In a decoding example, it requires 14.1 GB of XORs, and with Code-Specific Hybrid Reconstruction, that number is reduced to 12.6. *Luby*'s implementation does not optimize the encoding matrix, and therefore the penalty of decoding is not as great.



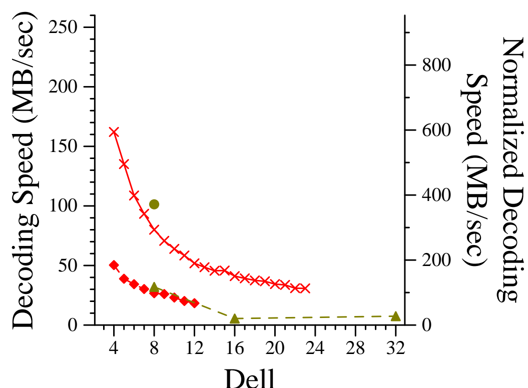


Figure 14: Decoding performance for [12,4].

As with the [6,2] tests, the performance of RS coding remains identical to decoding.

6 XOR Units

This section is somewhat obvious, but it does bear mentioning that the unit of XOR used by the encoding/decoding software should match the largest possible XOR unit of the machine. For exmaple, on the MacBook, the **long** and **int** types are both four bytes, while the **char** and **short** types are one and two bytes respectively. To illustrate the impact of word size selection for XOR operations, we test RDP performance for the [6,2] configuration on the MacBook with $w = 6$. The results in Figure 15 are expected.

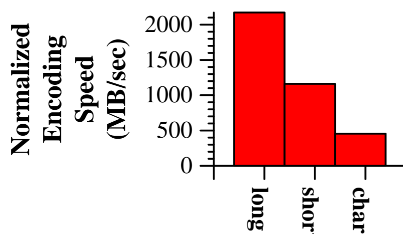


Figure 15: Effect of changing the XOR unit of RDP encoding when $w = 6$ in the [6,2] configuration on the MacBook.

The performance penalty at each successively smaller word size is more than a factor of two, since not only are twice as many XORs being performed, but the smaller data types must be extracted from their larger counterparts for each operation. All the libraries tested in this paper perform XORs with the widest word possible. This also suggests that 64-bit machines will yield significant improvements.

7 Conclusions

Given the speeds of current disks, the libraries explored here perform at rates that are easily fast enough to build high performance, reliable storage systems. We offer the following lessons learned from our exploration and experimentation:

RAID-6: The three RAID-6 codes, plus *Jerasure's* implementation of CRS coding for RAID-6, all perform much faster than the general-purpose codes. Attention must be paid to the selection of w for these codes: for RDP and EVENODD, it should be as low as possible; for Minimal Density codes, it should be as high as the caching behavior allows, and for CRS, it should be selected so that the primitive polynomial has a minimal number of ones. Note that $w \in \{8, 16, 32\}$ are all bad for CRS coding.

CRS vs. RS: For non-RAID-6 applications, CRS coding performs much better than RS coding, but now w should be chosen to be as small as possible, and attention should be paid to reduce the number of ones in the generator matrix. Additionally, a dense matrix representation should not be used for the generator matrix while encoding and decoding.

Parameter Selection: In addition to w, the packet sizes of the codes should be chosen to yield good cache behavior. To achieve an ideal packet size, experimentation is important; although there is a balance point between too small and too large, some packet sizes perform poorly due to direct-mapped cache behavior, and therefore finding an ideal packet size takes more effort than executing a simple binary search. Instead it may be semi-automated by using the region-based search of Section 4.4.

Minimizing the Memory Footprint: On some machines, the implementation must pay attention to memory. For example, *Jerasure's* RS implementation performs poorly on the Dell when $w = 16$ because it is wasteful of memory, while on the MacBook its memory usage does not penalize as much. Part of *Zfec's* better performance comes from its smaller memory footprint. In a similar vein, although we have not yet explored the notion completely, we have seen improvements in the performance of the XOR codes by reordering the XOR operations to minimize cache replacements. We anticipate further performance gains through this technique.

Beyond RAID-6: The place where future research will have the biggest impact is for larger values of m. The RAID-6 codes are extremely successful in delivering higher performance than their general-purpose counterparts. More research needs to be performed on special-purpose codes beyond RAID-6, and implementations need to take advantage of the special-purpose codes that already exist [9, 10, 16].

References

[1] Allmydata. Unlimited online backup, storage, and sharing. `http://allmydata.com`, 2008.

[2] Anvin, H. P. The mathematics of RAID-6. `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`, 2007.

[3] Beck *et al*, M. Logistical computing and internetworking: Middleware for the use of storage

in communication. In *Third Anual International Workshop on Active Middleware Services (AMS)* (San Francisco, August 2001).

[4] Blaum, M., Brady, J., Bruck, J., and Menon, J. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computing 44*, 2 (February 1995), 192– 202.

[5] Blaum, M., and Roth, R. M. On lowest density MDS codes. *IEEE Transactions on Information Theory 45*, 1 (January 1999), 46–59.

[6] Blomer, J., Kalfane, M., Karpinski, M., Karp, R., Luby, M., and Zuckerman, D. An XOR-based erasure-resilient coding scheme. Tech. Rep. TR-95-048, International Computer Science Institute, August 1995.

[7] Cleversafe, Inc. Cleversafe Dispersed Storage. Open source code distribution: `http://www.cleversafe.org/downloads`, 2008.

[8] Corbett, P., English, B., Goel, A., Grcanac, T., Kleiman, S., Leong, J., and Sankar, S. Row diagonal parity for double disk failure correction. In *4th Usenix Conference on File and Storage Technologies* (San Francisco, CA, March 2004).

[9] Feng, G., Deng, R., Bao, F., and Shen, J. New efficient MDS array codes for RAID Part I: Reed-Solomon-like codes for tolerating three disk failures. *IEEE Transactions on Computers 54*, 9 (September 2005), 1071–1080.

[10] Feng, G., Deng, R., Bao, F., and Shen, J. New efficient MDS array codes for RAID Part II: Rabin-like codes for tolerating multiple (≥ 4) disk failures. *IEEE Transactions on Computers 54*, 12 (Decemeber 2005), 1473–1483.

[11] Hafner, J. L. WEAVER Codes: Highly fault tolerant erasure codes for storage systems. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 211–224.

[12] Hafner, J. L. HoVer erasure codes for disk arrays. In *DSN-2006: The International Conference on Dependable Systems and Networks* (Philadelphia, June 2006), IEEE.

[13] Hafner, J. L., Deenadhayalan, V., Rao, K. K., and Tomlin, A. Matrix methods for lost data reconstruction in erasure codes. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 183–196.

[14] Huang, C., Chen, M., and Li, J. Pyramid codes: Flexible schemes to trade space for access efficienty in reliable data storage systems. In *NCA-07: 6th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2007).

[15] Huang, C., Li, J., and Chen, M. On optimizing XOR-based codes for fault-tolerant storage applications. In *ITW'07, Information Theory Workshop* (Tahoe City, CA, September 2007), IEEE, pp. 218–223.

[16] Huang, C., and Xu, L. STAR: An efficient coding scheme for correcting triple storage node failures. In *FAST-2005: 4th Usenix Conference on File and Storage Technologies* (San Francisco, December 2005), pp. 197–210.

[17] Karn, P. Dsp and fec library. `http://www.ka9q.net/code/fec/`, 2007.

[18] Luby, M. Code for Cauchy Reed-Solomon coding. Uuencoded tar file: `http://www.icsi.berkeley.edu/~luby/cauchy.tar.uu`, 1997.

[19] MacWilliams, F. J., and Sloane, N. J. A. *The Theory of Error-Correcting Codes, Part I.* North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[20] Nisbet, B. FAS storage systems: Laying the foundation for application availability. Network Appliance white paper: `http://www.netapp.com/us/library/analyst-reports/ar1056.html`, February 2008.

[21] Partow, A. Schifra Reed-Solomon ECC Library. Open source code distribution: `http://www.schifra.com/downloads.html`, 2000-2007.

[22] Plank, J. S. Jerasure: A library in C/C++ facilitating erasure coding for storage applications. Tech. Rep. CS-07-603, University of Tennessee, September 2007.

[23] PLANK, J. S. A new minimum density RAID-6 code with a word size of eight. In *NCA-08: 7th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2008).

[24] PLANK, J. S. The RAID-6 Liberation codes. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 97–110.

[25] PLANK, J. S., AND XU, L. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *NCA-06: 5th IEEE International Symposium on Network Computing Applications* (Cambridge, MA, July 2006).

[26] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8* (1960), 300–304.

[27] RHEA, S., WELLS, C., EATON, P., GEELS, D., ZHAO, B., WEATHERSPOON, H., AND KUBIATOWICZ, J. Maintenance-free global data storage. *IEEE Internet Computing 5*, 5 (2001), 40–49.

[28] RIZZO, L. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review 27*, 2 (1997), 24–36.

[29] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 1–16.

[30] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., MUELLER, B., SMALL, J., ZELENKA, J., AND ZHOU, B. Scalable performance of the Panasas parallel file system. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 17–33.

[31] WILCOX-O'HEARN, Z. Zfec 1.4.0. Open source code distribution: `http://pypi.python.org/pypi/zfec`, 2008.

[32] WYLIE, J. J., AND SWAMINATHAN, R. Determining fault tolerance of XOR-based erasure codes efficiently. In *DSN-2007: The International Conference on Dependable Systems and Networks* (Edinburgh, Scotland, June 2007), IEEE.

[33] XU, L., AND BRUCK, J. X-Code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory 45*, 1 (January 1999), 272–276.

[34] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 269–282.